

Homework 1:

Dei test ben progettati consentono di ridurre le possibilità che si verifichino dei bug all'interno del programma, con tutte le conseguenze che questi portano con sé. Un bug è sempre il risultato di un errore umano durante la scrittura del codice.

Il nostro obiettivo è ottenere un codice ben testato, ossia un codice che restituisca, nel maggior numero di casi possibile, i risultati attesi.

Caso di studio

Nel nostro caso, la progettazione dei test riguarda una classe java 'Automobile' da un programma per la gestione dei veicoli in un garage.

```
7 usages  ▲ Luke-cyb*  
public Automobile(String targa, String nome, String modello, int numeroPorte, int giornoImmatricolazione, int meseImmatricolazione,  
                  int annoImmatricolazione) {  
    if((((targa.matches(extorgo)) && (isDateValid(giornoImmatricolazione, meseImmatricolazione, annoImmatricolazione)))  
        &&(isModelValid(numeroPorte, modello)))&&(isDateMinor(giornoImmatricolazione, meseImmatricolazione, annoImmatricolazione))))  
    {  
        this.targa = targa;  
        this.nome = nome;  
        this.modello = modello;  
        this.numeroPorte = numeroPorte;  
        this.giornoImmatricolazione = giornoImmatricolazione;  
        this.meseImmatricolazione = meseImmatricolazione;  
        this.annoImmatricolazione = annoImmatricolazione;  
        System.out.println("Auto creata correttamente");  
    }else{  
        System.out.println("Dati non corretti");  
    }  
}
```

Abbiamo scritto tre metodi booleani che verificano che i parametri inseriti dall'utente rispettino certe condizioni, in modo da creare l'oggetto Automobile correttamente:

-**isDateValid(giorno, mese, anno)**: verifica che i parametri inseriti rappresentino una data realmente esistente.

-**isModelValid(numPorte, modello)**: verifica, in base al numero di porte e il modello inserito, che il tipo di auto che si vuole inserire sia corretto.

-isDateMinor(giorno, mese, anno): verifica che la data inserita sia minore o uguale alla data attuale. Non è possibile inserire una data futura.

Il fallimento di almeno uno di questi metodi non consente la creazione dell'oggetto Automobile.

Test:

I metodi possiedono tutti più di un parametro. Spesso la combinazione di valori errati con valori corretti può generare bug o a non far funzionare un metodo.

isDateValid: controlla che i parametri in input, giorno, mese, anno, formino una data realmente esistente.

Restituisce un valore booleano true o false.

```
public boolean isDateValid(int giorno, int mese, int anno) {  
    if ((anno >= 1900) && (anno <= Calendar.getInstance().get(Calendar.YEAR))) {  
        if ((mese >= 1) && (mese <= 12)) {  
            //controllo giorni  
            if ((giorno >= 1 && giorno <= 31) && (mese == 1 || mese == 3 || mese == 5 || mese == 7 ||  
                mese == 8 || mese == 10 || mese == 12)) {  
                return true;  
            }  
            else if ((giorno >= 1 && giorno <= 30) && (mese == 4 || mese == 6 || mese == 9 || mese == 11)){  
                return true;  
            }  
            else if ((giorno >= 1 && giorno <= 28) && (mese == 2)){  
                return true;  
            }  
            else if (giorno == 29 && mese == 2 && (anno % 400 == 0 || (anno % 4 == 0 && anno % 100 != 0))){  
                return true;  
            }  
            else {  
                return false;  
            }  
        } else {  
            return false;  
        }  
    } else {  
        return false;  
    }  
}
```

Variabile	Individual Input
	1<=giorno<=31

Giorno	1<=giorno<=30
	1<=giorno<=28
	1<=giorno<=29
	giorno<1
	giorno>31
	giorno>30
	giorno>28
	giorno>29
Mese	1<=mese<=12
	mese<1
	mese>12
Anno	1900<=anno<=anno corrente (2023)
	anno<1900
	anno>anno corrente

Combination of Input: c'è solo un'unica combinazione che è giorno - mese - anno.

Devise Test Cases: combinando tutti i possibili input, otteniamo: $9 \times 3 \times 3 = 81$.

-dayOutOfRange: questo test è stato pensato per capire come risponderebbe il programma ai valori della variabile giorno in relazione ai valori delle variabili mese e anno.

T1: giorno è 32 in un mese con massimo 31 giorni;

T2: giorno è 31 in un mese con un massimo di 30 giorni;

T3: giorno è 30 in un mese con un massimo di 28/29 giorni;

T4: giorno è 29 in un anno in cui mese ha massimo 28 giorni;

T5: giorno è 0 in un sistema in cui il giorno deve essere minimo 1;

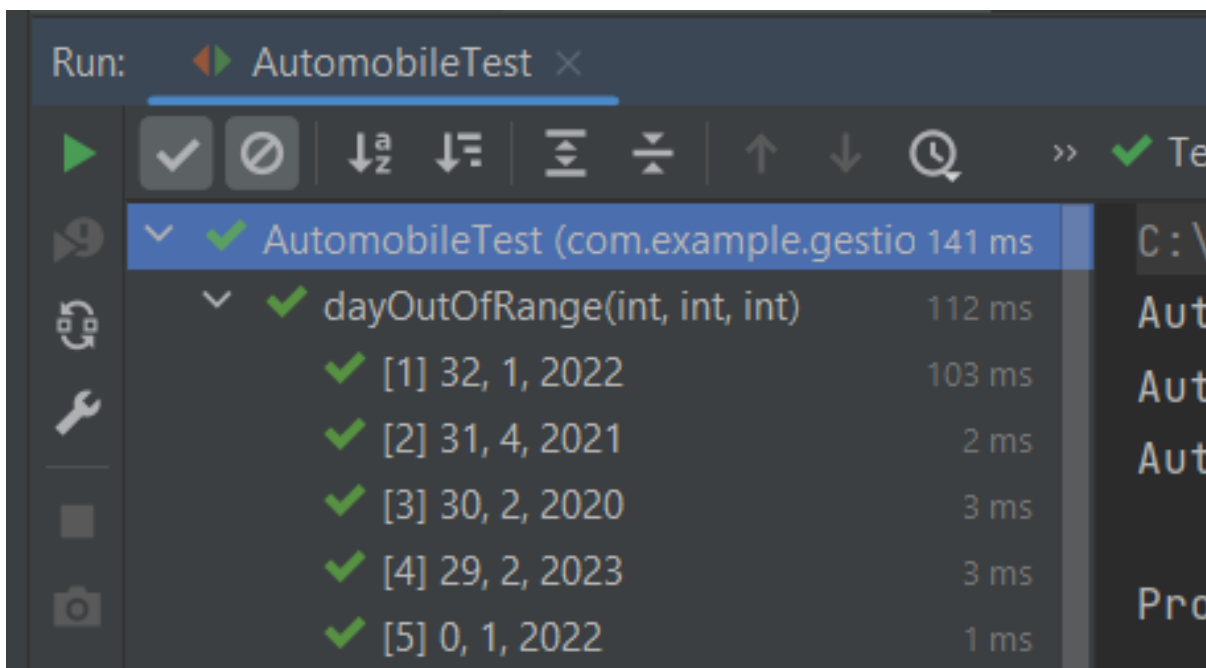
```

@ParameterizedTest
@CsvSource({
    "32, 1, 2022",
    "31, 4, 2021",
    "30, 2, 2020",
    "29, 2, 2023",
    "0, 1, 2022"
})
void dayOutOfRange(int giorno, int mese, int anno){

    assertFalse(auto.isDateValid(giorno, mese, anno));
}

```

Ci aspettiamo che il metodo `isDateValid` chiamato nell'asserzione restituisca, in tutti i casi selezionati, un valore booleano **false** e quindi che tutte le asserzioni `AssertFalse`, restituiscano **true**.



Tutti i test sono stati superati correttamente

-`monthOutOfRange`: questo test è stato pensato per capire come risponderebbe il programma ai valori della variabile mese in relazione ai valori delle variabili giorno e anno.

T1: mese è 13 in un sistema che prevede massimo 12 mesi;

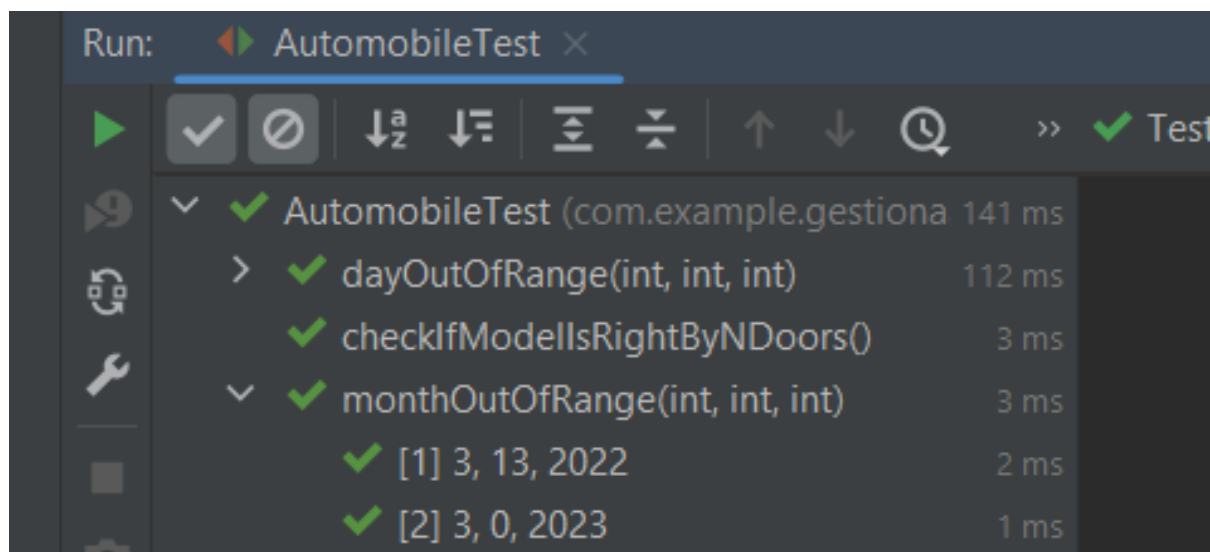
T2: mese è 0 in un sistema che prevede minimo 1 mese;

```

@ParameterizedTest
@CsvSource({
    "3, 13, 2022",
    "3, 0, 2023"
})
void monthOutOfRange(int giorno, int mese, int anno){
    assertFalse(auto.isDateValid(giorno, mese, anno));
}

```

I valori attesi dalle asserzioni sono gli stessi del test precedente.



Tutti i test sono stati superati correttamente

-yearOutOfRange: questo test è stato pensato per capire come risponderebbe il programma ai valori della variabile anno in relazione ai valori delle variabili giorno e mese.

T1: anno è 2024 in un periodo in cui l'anno corrente è 2023;

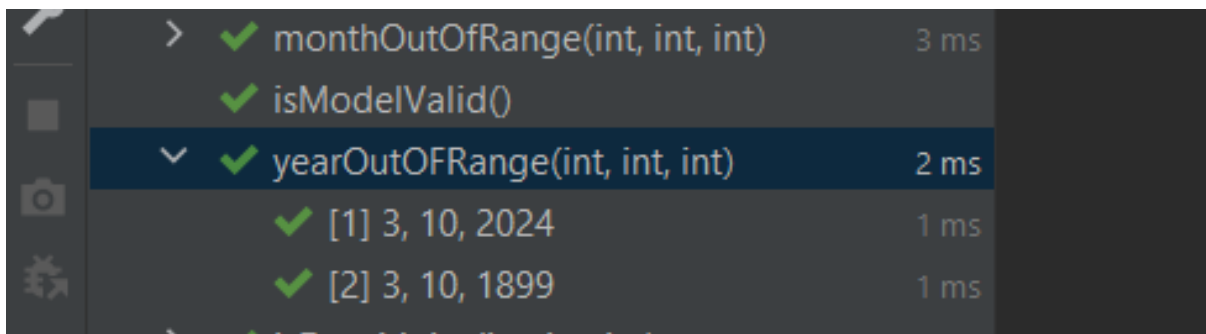
T2: anno è 1899 in un sistema che prevede che anno sia minimo 1900;

```

@ParameterizedTest
@CsvSource({
    "3, 10, 2024",
    "3, 10, 1899"
})
void yearOutOfRange(int giorno, int mese, int anno){
    assertFalse(auto.isDateValid(giorno, mese, anno));
}

```

I valori attesi dalle asserzioni sono gli stessi del test precedente.



>	✓	monthOutOfRange(int, int, int)	3 ms
	✓	isModelValid()	
✓	✓	yearOutOfRange(int, int, int)	2 ms
	✓	[1] 3, 10, 2024	1 ms
	✓	[2] 3, 10, 1899	1 ms

Tutti i test sono stati superati correttamente

-checkIfDatelsValid: questo test è stato pensato per capire come risponderebbe il programma ai valori limite imposti alle variabili giorno, mese, anno.

T1: giorno ha valore massimo (31), con mese valore minimo (1) e anno valore minimo (1900);

T2: giorno ha valore massimo diverso dal caso precedente (30), con mese che prevede quel valore di massimo (4) con anno valore massimo (2023);

T3: giorno ha valore massimo diverso dai casi precedenti (29), con mese (2) e anno (2020) che prevedono che giorno abbia quel valore massimo (anno bisestile);

T4: giorno ha valore massimo diverso dai casi precedenti (28), con mese (2) e anno (2022) che prevedono che giorno abbia quel valore massimo;

T5: giorno ha valore minimo (1), con mese valore massimo (12);

```

@ParameterizedTest
@CsvSource({
    "31, 1, 1900",
    "30, 4, 2023",
    "29, 2, 2020",
    "28, 2, 2022",
    "1, 12, 2000"
})
void isDateValid(int giorno, int mese, int anno) {
    assertTrue(auto.isDateValid(giorno, mese, anno));
}

```

Ci aspettiamo che il metodo `isDateValid` chiamato nell'asserzione restituisca, in tutti i casi selezionati, un valore booleano **true** e quindi che tutte le asserzioni `AssertTrue`, restituiscano **true**.

isDateValid(int, int, int)	7 ms
✓ [1] 31, 1, 1900	2 ms
✓ [2] 30, 4, 2023	2 ms
✓ [3] 29, 2, 2020	1 ms
✓ [4] 28, 2, 2022	1 ms
✓ [5] 1, 12, 2000	1 ms

Tutti i test sono stati superati correttamente

IsDateMinor: controlla che i parametri in input giorno, mese, anno, formino una data minore o uguale alla data corrente.

```

public boolean isDateMinor(int giornoImm, int meseImm, int annoImm){

    LocalDate dataInserita = LocalDate.of(annoImm, meseImm, giornoImm);
    LocalDate dataAttuale = LocalDate.now();

    return dataInserita.isBefore(dataAttuale);

}

```

-checkIfDatelsMinor: questo test è stato pensato per capire come risponderebbe il programma se le variabili giorno, mese, anno assumessero valori che formino una data minore di quella corrente.

T1: giorno, mese e anno, assumono l'ultimo valore corretto rispetto alla data di progettazione del test (11-05-2023)

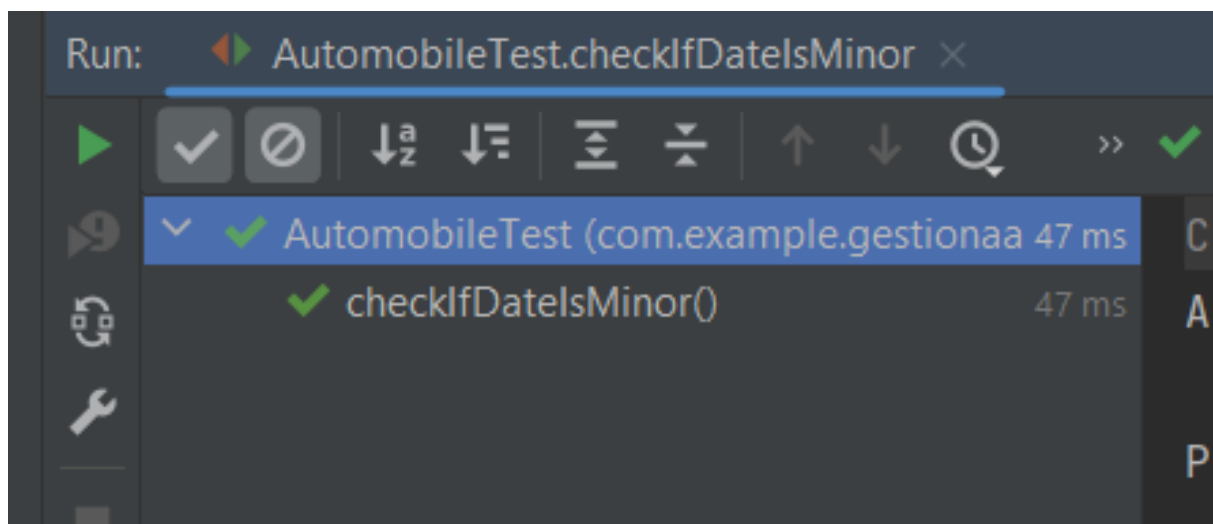
```

@Test
void checkIfDateIsMinor() {
    assertTrue(auto.isDateMinor( giornoImm: 11, meseImm: 5, annoImm: 2023));
}

```

no usages Luke-cyb *

Ci aspettiamo che il metodo isDateMinor chiamato nell'asserzione restituisca un valore booleano **true** e quindi che l'asserzione AssertTrue, restituisca **true**



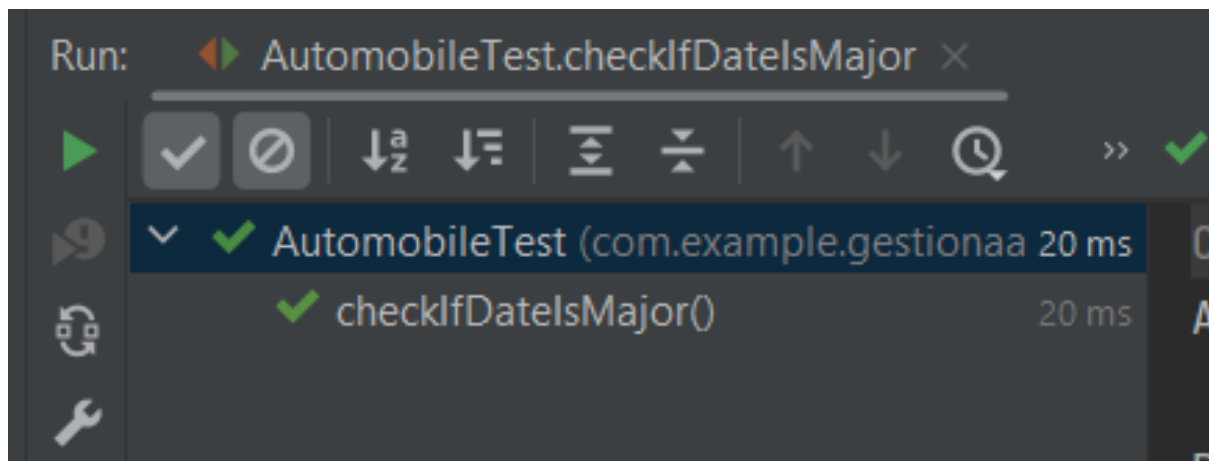
Il test è stato superato correttamente

-checkIfDatelsMajor: questo test è stato pensato per capire come risponderebbe il programma se le variabili giorno, mese, anno assumessero valori che formino una data maggiore di quella corrente.

T1: giorno, mese e anno, assumono l'ultimo valore non corretto rispetto alla data di progettazione del test (11-05-2023)

```
@Test
void checkIfDateIsMajor() {
    assertFalse(auto.isDateMinor( giornolmm: 3, meseImm: 11, annolmm: 2023));
}
```

Ci aspettiamo che il metodo `isDateMinor` chiamato nell'asserzione restituisca un valore booleano **false** e quindi che l'asserzione `AssertFalse`, restituisca **true**.



Il test è stato superato correttamente

isModelValid: controlla che il parametro modello sia corretto in base al valore del parametro numPorte.

```
public boolean isModelValid(int numeroPorte, String modello){
    boolean contains = false;
    for(int i=0; i<modelli.length; i++){
        if(modelli[i].toUpperCase().equals(modello.toUpperCase())){
            contains = true;
            break;
        }
    }
    if(contains){
        if(numeroPorte==2 && modello.toUpperCase().equals("COMPACT")) {
            return true;
        }else if(numeroPorte==3 && (modello.toUpperCase().equals("COMPACT") || modello.toUpperCase().equals("COUPE"))){
            return true;
        }else if(numeroPorte==5 && (modello.toUpperCase().equals("BERLINA") || modello.toUpperCase().equals("SUV"))){
            return true;
        }else{
            return false;
        }
    }
    return false;
}
```

Variabili	Individual Input
numPorte	numPorte = 2
	numPorte = 3
	numPorte = 5
	numPorte = altri numeri
modello	Null
	Empty
	modello = "Compact"
	modello = "Coupé"
	modello = "Berlina"
	modello = "Suv"
	modello = Altre stringhe

Combination of Input:

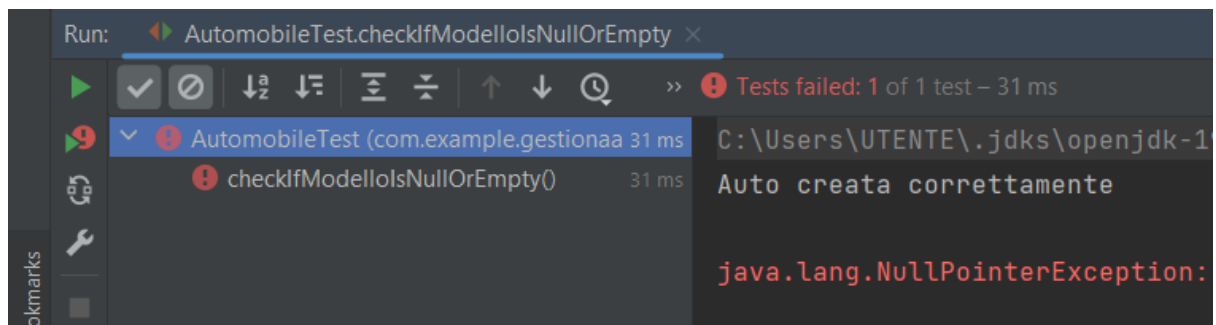
- modello può essere "Compact" se numPorte = 2 o numPorte = 3;
- modello può essere "Coupé" se numPorte = 3;
- modello può essere "Berlina" o "Suv" se numPorte = 5;

Devise Test Cases: $4 \times 7 = 28$.

-checkIfModelloIsNullOrEmpty: questo test è stato pensato per capire come risponderebbe il programma se il parametro modello dovesse essere una stringa null o vuota.

```
@Test
void checkIfModelloIsNullOrEmpty(){
    assertFalse(auto.isModelValid( numeroPorte: 2,  modello: null));
    assertFalse(auto.isModelValid( numeroPorte: 2,  modello: ""));
}
```

Ci aspettiamo che il metodo isModelValid chiamato nelle asserzioni restituisca un valore booleano **false** e quindi che le asserzioni AssertFalse, restituiscano **true**.



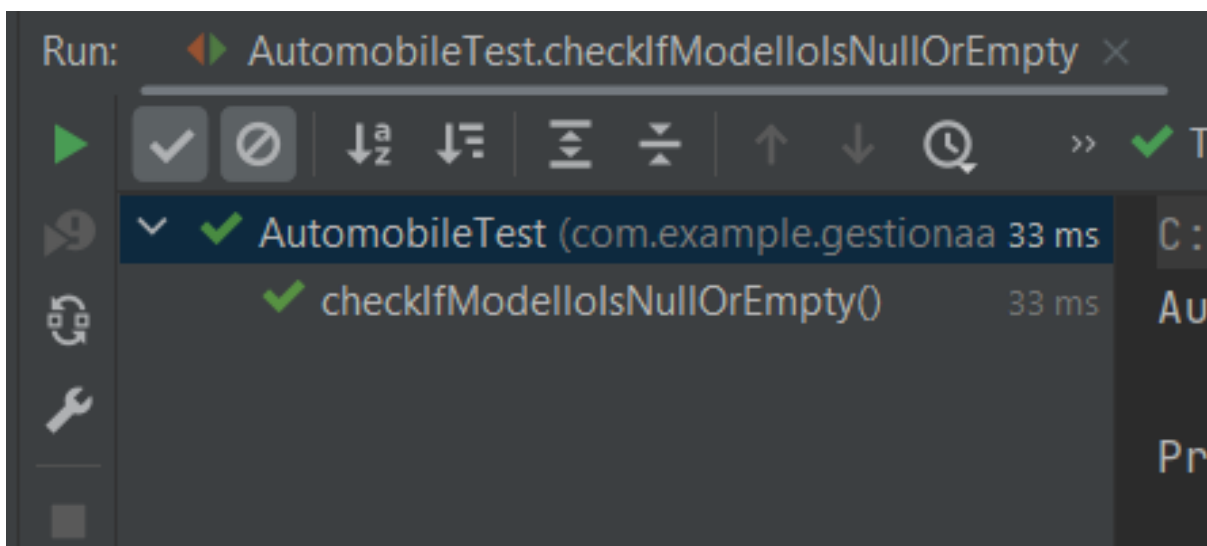
Il test fallisce. Stando ai risultati, il codice non prevede il controllo sulla stringa modello per questo input.

```

public boolean isModelValid(int numeroPorte, String modello) {
    boolean contains = false;
    if (modello != null) {
        for (int i = 0; i < modelli.length; i++) {
            if (modelli[i].toUpperCase().equals(modello.toUpperCase())) {
                contains = true;
                break;
            }
        }
        if (contains) {
            if (numeroPorte == 2 && modello.toUpperCase().equals("COMPACT")) {
                return true;
            } else if (numeroPorte == 3 && (modello.toUpperCase().equals("COMPACT") || modello.toUpperCase().equals("COUPE"))) {
                return true;
            } else if (numeroPorte == 5 && (modello.toUpperCase().equals("BERLINA") || modello.toUpperCase().equals("SUV"))) {
                return true;
            } else {
                return false;
            }
        }
    }
    return false;
} else {
    return false;
}
}

```

Il metodo è stato corretto con l'inserimento del controllo sul valore null.



Il test è stato superato correttamente

-checkIfModellsRightByNDdoors: questo test è stato pensato per capire come risponderebbe il programma se il parametro modello è compatibile col valore di numPorte.

```

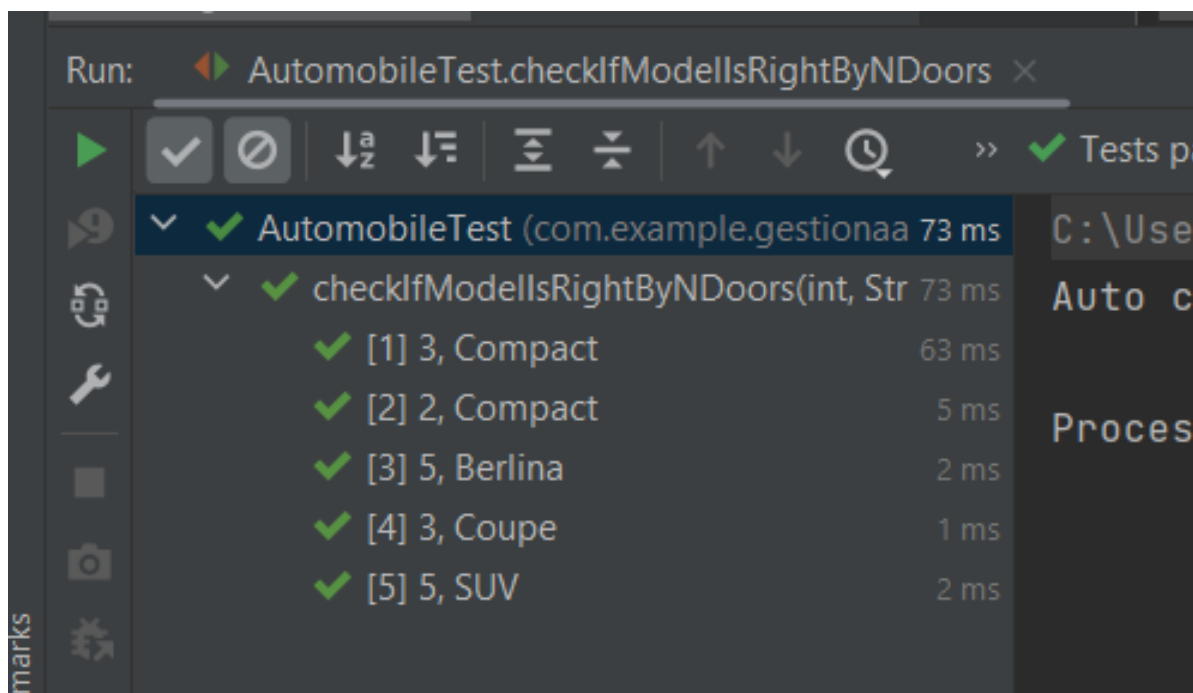
@ParameterizedTest
@CsvSource({
    "3, Compact",
    "2, Compact",
    "5, Berlina",
    "3, Coupe",
    "5, SUV"
})
void checkIfModelIsRightByNDdoors(int numPorte, String modello){

    assertTrue(auto.isModelValid(numPorte, modello));

}

```

Ci aspettiamo che il metodo `isModelValid` chiamato nell'asserzione restituisca, in tutti i casi selezionati, un valore booleano **true** e quindi che tutte le asserzioni `AssertTrue`, restituiscano **true**.



Tutti i test sono stati superati correttamente

-`checkIfModellsRightByNDdoors`: questo test è stato pensato per capire come risponderebbe il programma se dovesse ricevere un modello incompatibile con numPorte, e se quest'ultimo non fosse un valore previsto nel set di input.

```

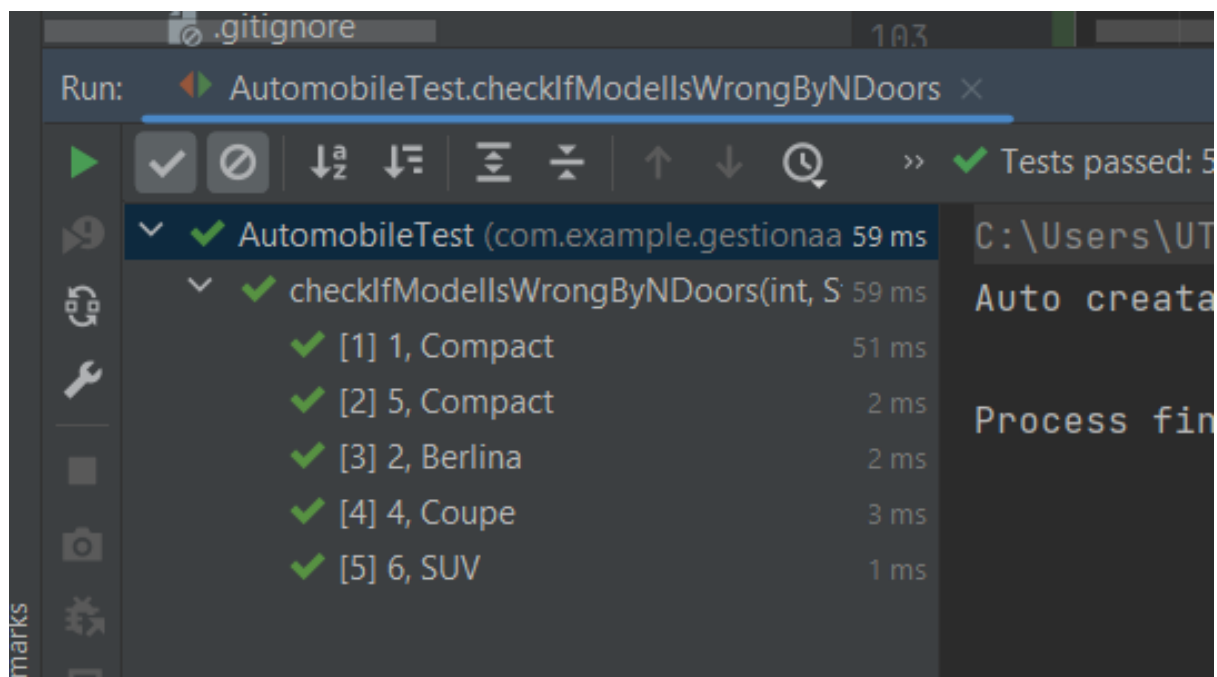
@ParameterizedTest
@CsvSource({
    "1, Compact",
    "5, Compact",
    "2, Berlina",
    "4, Coupe",
    "6, SUV"
})
void checkIfModelIsWrongByNDoors(int numPorte, String modello){

    assertFalse(auto.isModelValid(numPorte, modello));

}

```

Ci aspettiamo che il metodo `isModelValid` chiamato nell'asserzione restituisca, in tutti i casi selezionati, un valore booleano **false** e quindi che l'asserzione `AssertFalse`, restituisca **true**



Tutti i test sono stati superati correttamente

TestCases

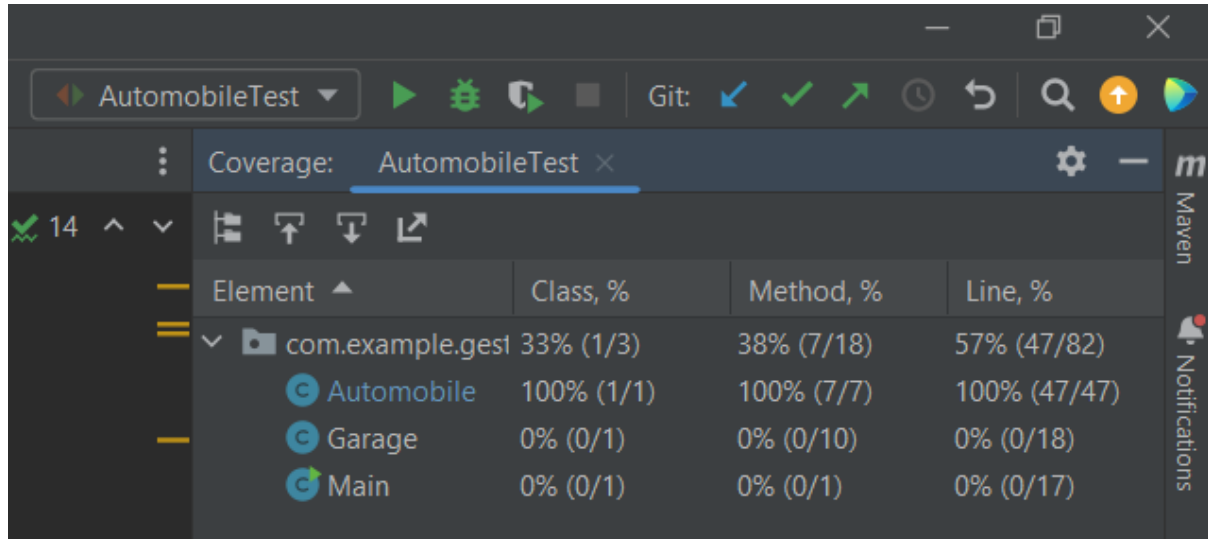
Homework 2:

Task n1:

La suite di test progettata in black box si è rivelata sufficiente.

Code Coverage:

L'analisi code coverage sulla suite di test mostra una copertura del 100% del codice.



Element	Class, %	Method, %	Line, %
com.example.gest	33% (1/3)	38% (7/18)	57% (47/82)
Automobile	100% (1/1)	100% (7/7)	100% (47/47)
Garage	0% (0/1)	0% (0/10)	0% (0/18)
Main	0% (0/1)	0% (0/1)	0% (0/17)

Task n2:

swapLetters(): Il metodo scambia due lettere all'interno di una stringa. Dati due interi, corrispondenti alle posizioni nella stringa, il metodo scambia le due lettere nelle rispettive posizioni. Se i due interi sono maggiori della lunghezza della stringa, questa viene estesa con delle x a sinistra e a destra per rendere possibile lo scambio.

```
2 usages
5  @
6  public static String swapLetters(int pos1, int pos2, String inputString, int numMax) {
7
8      if (pos1 <= 0 || pos2 <= 0 || pos1 >= pos2) {
9          return null;
10      }
11
12      int stringLength = inputString.length();
13
14      if ((pos1 > stringLength || pos2 > stringLength) && (pos2 <= numMax)) {
15          int leftPadding = pos1 - stringLength; // leftPadding = 2
16          int rightPadding = pos2 - stringLength; // rightPadding = 3
17
18          StringBuilder stringBuilder = new StringBuilder();
19
20          //aggiunge x a sinistra
21          for (int i = 0; i < leftPadding; i++) {
22              stringBuilder.append("x");
23          }
24
25          //aggiunge x a destra
26          stringBuilder.append(inputString);
27      }
28  }
```

```

24 | //aggiugne x a destra
25 | stringBuilder.append(inputString);
26 |
27 | for (int i = 0; i < rightPadding; i++) {
28 |     stringBuilder.append("x");
29 | }
30 | //inputString = xxabcdxxx
31 | inputString = stringBuilder.toString();
32 |
33 | }
34 |
35 | char[] charArray = inputString.toCharArray();
36 |
37 | char temp = charArray[pos1-1]; //posizioni decrementate per la gestione dell'array
38 | charArray[pos1-1] = charArray[pos2-1];
39 | charArray[pos2-1] = temp;
40 |
41 | // Stringa restituita: xxabxcdxx
42 | return new String(charArray);
43 | }
44 |
45 | }
46 |

```

T1: Il valore di pos1 o pos2, variabile che rappresenta la posizione all'interno dell'array che si vuole scambiare.

```

@Test
void PosMinorThanZero(){
    assertNull(Swaps.swapLetters( pos1: -1, pos2: 2, inputString: "Ciao", numMax: 5));
}

```

Il valore di pos1 è minore di 0, le altre variabili assumono valori validi. Il metodo prevede di ritornare **null**.

✓ PosMinorThanZero()

Il test è stato superato correttamente.

T2: I valori di pos1 e pos2 sono più grandi della dimensione della stringa in ingresso e più piccoli di numMax (un valore numerico arbitrario).

```

@Test
void PosOverTheLimit(){
    assertEquals( expected: "xxcanxexx", Swaps.swapLetters( pos1: 6, pos2: 7, inputString: "cane", numMax: 10));
}

```


Il metodo espande la stringa con delle 'x' in base alle differenze di pos1 e pos2 con la dimensione della stringa. In questo caso ci aspettiamo un'espansione a sinistra di due x e a destra di tre. La stringa viene elaborata successivamente. Il metodo, con input **cane**, prevede di ritornare la stringa **xxcanxexx**.

✓ PosOverTheLimit()

Il test viene passato correttamente.

Il criterio di code coverage seguito è il **Line Coverage**: una linea di codice è coperta se viene eseguita almeno una volta.

Abbiamo identificato 4 linee principali da coprire: due istruzioni if e due istruzioni for, queste ultime contenute nel secondo if.

Ora dimostreremo come i test realizzati non sono sufficienti a coprire il codice, nella prospettiva di un'analisi black box .

Test Black Box

Variabili	Individual input
pos1	pos1>0
	pos1<=0
pos2	pos2>0
	pos2<=0
InputString	null
	empty
	stringa qualunque
numMax	numMax>0
	numMax<=0

Combinazioni di input:

(BUONI)

c1. $\text{pos1} < \text{pos2}$, $\text{pos2} \leq \text{numMax}$, $\text{pos2} < \text{InputString.length()}$, $\text{InputString.length()} \leq \text{numMax}$;

esempio: `swapLetters(1,3,"aico",4) ⇒ "ciao"`;

c2. $\text{pos1} < \text{pos2}$, $\text{pos2} \leq \text{numMax}$, $\text{pos2} > \text{InputString.length()}$, $\text{InputString.length()} \leq \text{numMax}$;

La posizione 2 è più lunga della stringa e vengono aggiunte le x a destra.

esempio: `swapLetters(1,7,"aico",10) ⇒ "xiocxxa"`;

c3. $\text{pos1} < \text{pos2}$, $\text{pos2} \leq \text{numMax}$, $\text{pos1} > \text{InputString.length()}$, $\text{pos2} > \text{InputString.length()}$, $\text{InputString.length()} \leq \text{numMax}$;

La posizione 1 è più lunga della stringa ma è più piccola di pos 2 vengono aggiunte le x sia a sinistra che a destra.

esempio: `swapLetters(5,8,"aico",10) ⇒ "xciaxxa"`;

(CATTIVI)

c4. $\text{pos1} > \text{pos2}$;

la prima posizione non può essere più grande della prima

esempio: `swapLetters(8,3,"aico",20) ⇒ null`;

c5. $\text{InputString.length()} > \text{numMax}$;

Se il numero di caratteri della stringa inserita è più grande di numMax il metodo restituisce una stringa vuota

esempio: `swapLetters(5,6,"automobile",8) ⇒ empty`;

c6. $\text{pos1} > \text{numMax}$;

Se la 1^ posizione è più grande di numMax il metodo restituisce null perchè non può effettuare l'operazione

esempio: `swapLetters(5,9,"automobile",4) ⇒ null`;

c7. $\text{pos2} > \text{numMax}$;

Se la 2^ posizione è più grande di numMax il metodo restituisce null perchè non può effettuare l'operazione

esempio: `swapLetters(5, 9, "automobile", 6) ⇒ null`;

(combination of Individual input)*(Combinazioni di input)=Numero totale di test
(2x2x3x2)X(7)=168 test case;

TestCaseTemplateTask2

Homework 3

```
public class Coordinate {  
  
    1 usage  
    public static double calculateDistance(double x1, double y1, double x2, double y2) {  
        double deltaX = x2 - x1;  
        double deltaY = y2 - y1;  
  
        double distance = Math.sqrt(Math.pow(deltaX, 2) + Math.pow(deltaY, 2));  
  
        return distance;  
    }  
}
```

Nell'esempio sopra, abbiamo un property-based test denominato **testCalculateDistance** che verifica se il metodo **calculateDistance** contenuto nella classe "Coordinate" restituisce la distanza corretta tra due punti nel piano cartesiano.

```
class CoordinateTest {  
  
    no usages  
    @Property  
    @Report(Reporting.GENERATED)  
    void calculateDistance(@ForAll double x1, @ForAll double y1, @ForAll double x2, @ForAll double y2) {  
        double distance = Coordinate.calculateDistance(x1, y1, x2, y2);  
        double expectedDistance = calculateExpectedDistance(x1, y1, x2, y2);  
        assertEquals(expectedDistance, distance); // Tolleranza per errori di arrotondamento  
  
        Statistics.collect(x1, y1, x2, y2);  
    }  
  
    1 usage  
    private double calculateExpectedDistance(double x1, double y1, double x2, double y2) {  
        double deltaX = x2 - x1;  
        double deltaY = y2 - y1;  
        return Math.sqrt(Math.pow(deltaX, 2) + Math.pow(deltaY, 2));  
    }  
}
```

Utilizziamo `assertEquals` per confrontare il valore atteso “`expectedDistance`” con il risultato restituito dal metodo `calculateDistance`.

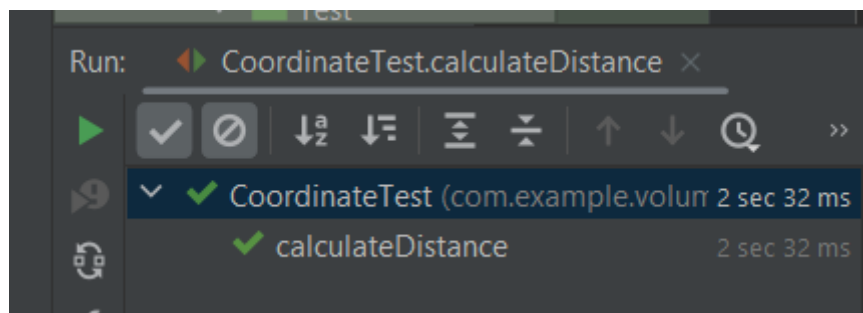
I parametri `x1`, `y1`, `x2` e `y2` rappresentano le coordinate dei due punti nel piano cartesiano e vengono generati casualmente. L'annotazione `@ForAll` indica che tutti i parametri devono essere generati casualmente.

Il metodo `calculateExpectedDistance` nella classe test calcola manualmente la distanza attesa tra i due punti utilizzando una formula analoga a quella utilizzata nel metodo `calculateDistance`. Questo ci consente di verificare il risultato restituito dal metodo `calculateDistance` confrontandolo con il valore atteso.

Questo tipo di test è utile perchè il metodo utilizza operazioni complesse, come la radice quadrata. Uno dei bug più comuni potrebbe nascere in seguito a errori di approssimazione dovuti all'arrotondamento eseguito dalla macchina.

```
timestamp = 2023-06-21T15:54:45.534591400, [CoordinateTest:calculateDistance] (1000) statistics =
0.0 0.0 0.0 0.0 (1) : 0.10 %
1.6653638826111794E100 1.4617497486230308E255 -6.126388666851781E16 -2.0203757523125316E272 (1) : 0.10 %
1.59 635.51 -9027.87 -5.848505654276292E236 (1) : 0.10 %
4.912018157664735E204 8.6136376626645E11 7.647323800537965E171 8.393248180110298E307 (1) : 0.10 %
-3.1113863858483045E272 -3.850312181224355E74 1.0371179515864282E205 6.565574040477128E113 (1) : 0.10 %
2.099170230732217E32 -22.66 1415.01 -14.26 (1) : 0.10 %
0.0 0.0 0.0 1.0 (1) : 0.10 %
-6.726155484540151E307 1.7646747032899207E238 0.01 100.6 (1) : 0.10 %
-10167.03 -2.1925245049189114E17 -0.1 8.286001023106461E62 (1) : 0.10 %
-7.63720615053379E23 -6.276004519015047E236 -7.6056007275004E11 7.054758354853005E188 (1) : 0.10 %
0.0 0.0 0.0 -1.0 (1) : 0.10 %
0.0 0.0 1.0 0.0 (1) : 0.10 %
150.33 7.4474500057/237/584 35.1 1.45108051037/8/085100 (1) : 0.10 %
```

Il test è stato eseguito con 1000 valori casuali diversi per ogni parametro (`x1`, `y1`, `x2`, `y2`)



Il test è stato superato correttamente per ogni serie di valori.